Replicated Database Recovery using Multicast Communication *

JoAnne Holliday  
Dept of Computer Engineering  
Santa Clara University,  
Santa Clara, CA 95053  
jholliday@acm.org

Abstract

Database replication with update-anywhere capability while maintaining global synchronization and isolation has long been thought impractical. Protocols have been proposed for distributed replicated databases that take advantage of atomic broadcast systems to simplify message passing and conflict resolution in hopes of making replication efficient. This paper presents global recovery algorithms to handle site failures when such protocols are used with a broadcast system providing virtual synchrony.

1 Introduction

Database replication has been a topic of interest for many years. Replication has improved performance, availability and fault tolerance for many distributed systems; however, these advantages have not been realized for replicated databases, which must contend with multi-item, multi-operation transactions with the ACID properties [4]. It is difficult to obtain good performance from database replication when updates can be processed at any site and one-copy serializability must be enforced. This has lead to research into primary-copy models that restrict where updates can occur and schemes which relax consistency, atomicity and isolation requirements. Distributed database researchers have explored the traditional read-one, write-all update protocols with Two Phase Locking (2PL) or Global Time-Stamp and variations which use quorums and lazy update propagation. The numerous difficulties and coping strategies are summarized in [16].

Several researchers have proposed using group broadcast (or multicast) communications for replica update to enforce determinism and alleviate some of the problems with “update everywhere” replication [1, 6, 7, 9, 10]. One reason these protocols have not been used in commercial systems is that some areas are not yet well defined. One of these areas is recovery. Most research to date on these multicast based protocols has assumed that the sites did not fail or they were fail-stop, that is, if they failed, they simply stopped and did not recover.

When recovery or the addition of new sites is considered, these sites update and re-synchronize their database copies by copying the entire database of another site [10, 11]. This consumes time and bandwidth, as normal update operations of the database system are interrupted. Replicated database systems can compensate for this by having a larger number of database servers than might be necessary for normal operations. This way, if some of the servers go down, it is possible to wait for a convenient time to interrupt the system and make it unavailable while those server sites are recovering. We propose a recovery mechanism for replica update protocols that are, according to the classification of Wiesmann et al. [15] update everywhere and non-voting. The goal of these recovery mechanisms is to minimize system down time and the disruption caused by failures.

The paper is organized as follows. Section 2 presents our system model. Section 3 presents three replica update protocols that can run on such a system. Section 4 presents global recovery methods for the three protocols. We conclude with Section 5.

*Dr. Holliday is the Clare Boothe Luce Professor of Computer Engineering at Santa Clara University
2 System Model and Assumptions

In this paper we consider a highly available database server in an Internet environment. If the server is implemented as a single database, it may easily become congested and become a bottleneck. We therefore replicate several copies of the database (or perhaps only the hot-spot pages) on multiple server sites connected by a local area network. We assume that users interact with the database by invoking transactions at any one of the server copies [4]. We consider only simple transactions that are a sequences of read and write operations on the data items that preserve the ACID properties (atomicity, consistency, isolation, durability). A transaction can originate at any server site, and that site becomes the initiating or home site. Concurrency control is locally enforced by the database management system (DBMS) at all server sites with strict two phase locking. However, this is not sufficient to ensure correctness in the presence of replication as a transaction can be initiated at any one of the database copies and an update protocol is needed to ensure one-copy serializability [4]. That is, the multi-copy database must appear to the users as a single copy database.

We assume the sites are connected by an efficient network with some sort of atomic multicast system providing virtual synchrony [12]. Several systems have been built that provide atomic broadcast for message passing among processes in a distributed system on a LAN, e.g., Totem [2] and Amoeba [8]. Other atomic broadcast systems have been proposed to provide these services on the Internet [13] and over wireless networks [3]. This communication subsystem connecting them provides primitives to broadcast messages to a group of sites. These group broadcast (or multicast) services ensure reliable totally ordered message delivery to the processes at the sites. In order to achieve the ordered delivery of messages, messages that are received at a site are delayed and not delivered to the application until they can be delivered in the proper order. In this paper we consider atomic broadcast with the following properties [5]: If a site broadcasts a message \( m \), the primitive ensures that the message will be delivered to all operational sites. Furthermore, if a site delivers a message \( m \), then all operational sites deliver \( m \). This is the “all-or-nothing” property. A message is delivered at most once, and only if it was actually broadcast. The messages have a total order. That is, if one site delivers the broadcast message \( m \) before \( m' \), then all sites deliver message \( m \) before \( m' \). In addition to providing message ordering and reliability, multicast systems keep track of group membership and report site failures.

Each site DBMS has a local recovery mechanism [4], so that transactions committed by the site are durable and will survive site failures (we assume a site failure is equivalent to a loss of power at the site and thus the contents of volatile memory are lost). In order to accomplish this, the database is kept on some type of stable storage and parts of the database are brought into cache or volatile memory as needed to be read or modified. When a transaction modifies data, that modified data must be either flushed to the stable database or a record of the update must be written to a recovery log on a non-volatile medium before the transaction can be committed. After a failure, the recovery log is used to update the stable database so that the effects of uncommitted transactions that may have been flushed to the stable database are undone and the effects of committed transactions that were not written to the stable database are applied. We call this local recovery. Global recovery refers to the process of bringing a site which has failed and gone through local recovery up-to-date with respect to the rest of the sites in the distributed database system. This is necessary if we allow the distributed system to continue to process updates even though one or more sites have failed. We do not restrict the number of potentially faulty sites, however, specific multicast systems may have this restriction. We assume that in the case of site or communication failures, only a primary group (identified by the group membership mechanism) continues to process database updates and any other subgroup knows that it is non-primary and thus does not process updates but waits to rejoin the primary group.

The communication system providing virtual synchrony keeps track of the group membership and broadcasts messages to members (sites) of the group. The set of sites that are currently operational is called the current view. Any time the failure or recovery of one of the member sites causes a change in the view, the communication system sends a view change message to each member. A property basic to most of these systems is that processes (sites) moving together from view \( v \) to view \( v' \) should deliver the same set of messages in view \( v \). Also that every message \( m \) be delivered in the same view \( v \) by all sites that deliver \( m \) [14]. The problem addressed in this paper is that since the multicast-based protocols have no 2 Phase Commit (or other atomic commit protocol), a transaction, \( t_1 \), might be committed by operational sites despite the failure of one of the replicas. When the failed site recovers, it has no knowledge of \( t_1 \) and thus has an
inconsistent copy of the database. The current global recovery mechanism of sending a complete copy of the current state of the database over the network while the system activity is suspended is inefficient and can lead to excessive system unavailability if the database is large.

3 Existing Replica Update Protocols Using Atomic Multicast

We now briefly describe three replicated data update protocols [1] which use atomic broadcast primitives as their means for communication with the other replicas and ensure one-copy serializability. These protocols were proposed in [1] and are summarized here. These protocols are based on reading one copy and writing all copies of replicated objects. The idea underlying each of these protocols is that since the communication subsystem guarantees totally ordered broadcasts, operations can be processed at every site in the same order. These protocols execute read operations locally and multicast only write operations. A read operation is executed locally after obtaining a read lock, and write operations are broadcast to all copies of the database. The database copies, therefore, do not go through the exact same states with respect to read locks, only with respect to write locks. Because of this, write operations take precedence over read operations. Since the atomic broadcast has the “all-or-nothing” property, there is no need to employ an atomic commitment protocol.

Atomic broadcast tools can be quite expensive in terms of message passing overhead as compared to point-to-point communication. Due to the order guarantees, multiple rounds of message passing may be needed by the multicast system. However, the replicated data protocols presented here exploit these total order guarantees to attain several advantages. When message delivery is guaranteed, there is no need to wait for acknowledgments for operation requests as is required in most point-to-point communication schemes. Since the state in which an operation is executed is identical at all sites, the response to all operations is the same. A two-phase commit protocol to ensure atomic commitment is not needed since all sites will uniformly make the same decisions. Elimination of explicit acknowledgments and two-phase commit at the application level should reduce the communication cost. Similar operations are performed by the multicast system at a lower level, however, the lower level operations should be more efficient. One of the main problems encountered when two-phase locking is used in a distributed environment is the need for distributed deadlock detection. In these protocols, global deadlock is impossible. These multicast-based protocols will cause some unnecessary transaction aborts. However, this drawback must be weighed against the advantage of not having to worry about deadlocks. One study [7] has indicated that the unnecessary aborts are not a serious disadvantage.

3.1 Broadcast Writes

In this protocol, a read operation is executed locally after obtaining a read lock, and a write operation is broadcast to all copies of the database. When the lock manager at site $S$ delivers a write request from $t_i$, it checks if the lock can be granted. If granting is successful, the operation is executed. If the lock cannot be granted, it is because the requested data page, $x$, is either locked by readers or a writer. The write operation takes precedence over readers, which are local transactions. This ensures that there can never be a deadlock cycle involving read operations. Since write operations are performed globally, a cycle involving write operations will be detected at every site in the system and resolved consistently. Finally, $t_i$ terminates by broadcasting a commit request and each site makes the same commit or abort decision when the request is delivered.

3.2 Delayed Broadcast

The “Delayed Broadcast” protocol [1] localizes transaction execution by deferring update operations until commit time, when a single message with all updates is sent to all other sites. A transaction, $t_i$, executes a read operation locally while each write operation is deferred until $t_i$ is complete. When $t_i$ has done all of its reads and calculations, $t_i$ broadcasts its deferred writes to all sites. On receiving the write message, the lock manager at site $S$ (this includes the home site) obtains write locks for $t_i$ as described above, and then the writes are executed at $S$. After all the writes of $t_i$ are executed
at the home site, $t_i$ broadcasts its commit operation to all sites. $t_i$ terminates after the delivery and execution of its commit request at each site.

Deadlocks are not possible with this protocol. This is because all the write locks for a transaction are requested in a single atomic step at the local lock manager thus preventing a cycle of write requests. Conflicts with read locks are dealt with by aborting the transaction with the read lock. There is thus no need to check for deadlocks.

### 3.3 Single Broadcast Protocol

Since all write operations are known to all sites and they will be eventually executed, the extra broadcast of the commit operation can be avoided. To use a single broadcast, the Single Broadcast protocol [1] maintains a version number for each page in the database. A transaction $t_i$ executes a read operation locally and a write operation is deferred until $t_i$ is ready to commit as in Delayed Broadcast. When the site that initiated $t_i$ is ready to commit, if $t_i$ is a read-only transaction, the decision to commit is done locally and no message is broadcast (also like Delayed Broadcast). For an update transaction, the site broadcasts the identity of the set of pages read with their version numbers and the set of pages to be written with their new values. On receiving the set of reads and writes of $t_i$, the lock manager at a site $S$ first checks if the version of the items read by $t_i$ are obsolete (that is, if any of the versions of the pages at $S$ are greater than the versions read by $t_i$). If so, $t_i$ is aborted. Otherwise, $S$ proceeds with the attempt to atomically grant all the write locks. If a write lock cannot be granted due to a conflict with a read operation, the reading transaction is aborted and $t_i$ receives the lock. Once all write locks are obtained, $S$ executes the write operations and increments the version numbers of each data-item. $t_i$ terminates at $S$ as soon as it obtains the write locks and successfully executes its write operations.

### 4 Failure and Recovery

The simplest protocol from a recovery standpoint is Single Broadcast and so the global recovery method for Single Broadcast will be presented first. The recovery algorithm for the Delayed Broadcast and Broadcast Writes protocols use the methods developed for Single Broadcast and add to them.

#### 4.1 Single Broadcast Recovery

When a multicast message is delivered to the database, the corresponding transaction is executed and committed or it is aborted. Since the termination request is implicit in the message, once the message is delivered to the database application, a subsequent failure of the site will not affect the disposition of the transaction. Let us examine what happens when a site fails. The multicast subsystem will detect a failure and its membership protocol will create a new view excluding the failed site. Operational sites will receive a view change message, say from $v$ to $v'$. If a transaction commit request message for $t_1$ was delivered in view $v$, then it was delivered by all sites that were part of $v$, and thus the transaction $t_1$ was committed (or aborted) by all sites in $v$. When the site that failed and was excluded in $v'$ (call it $S_i$) recovers locally, it will have the effects of transaction $t_1$ in its database. When $S_i$ re-joins the group, another view, $v''$ will be installed. Although $S_i$ has the effects of $t_1$ and all other transactions that committed in $v$, the effects of transactions that committed in $v'$ are not in its database. We could recover $S_i$ by doing a complete database transfer with some other operational site, however, this will probably be time consuming and if $S_i$ has only been off-line for a short time or the rate of updates is low, a complete data transfer is not needed. In this paper, we attempt to avoid doing a complete database transfer if possible since the processing of update transactions at all sites must be delayed during global recovery.

Some group communication systems with virtual synchrony provide a recovery mechanism for failed and recovered members. These mechanisms log the delivered messages so that when the member recovers, the missed messages can be replayed to the recovering site. Such a mechanism can be used for global recovery of database sites using the Single Broadcast protocol.
If the group communication system does not provide for global recovery, we designate some of the sites to be update messages **Loggers**. A Logger that is also a database site can log messages intelligently. For example, none of the messages having to do with transactions which ultimately were aborted need to be logged. Therefore the Logger, which is also a database site, logs view changes and transaction commit records. This message or change log is an ordered list of records and is different from the recovery log which is maintained by the DBMS [4] at each site and is used for local recovery. When a broadcast message is delivered and it indicates a view change, the Logger makes an entry in the change log indicating the membership changes. When a broadcast message is delivered indicating an update transaction, it is also added to the change log. If that transaction was found to have read stale data, the entry is removed and the transaction aborted. Otherwise, the transaction is executed and committed. The change log is used as follows: When the multicast communication system detects a membership change and a site is added to the view, no update transaction messages are delivered to it (or to any other site) until the new site has exchanged messages with one of the Logger sites and the Logger has signaled that global recovery is complete. The Logger sees a view change message followed by a request from the new site, $S_i$, to be brought up-to-date. The Logger looks up the last view of which $S_i$ was a member (or $S_i$ can tell the Logger and the Logger merely verifies this). If $S_i$ has been absent from the group for so long that the Logger no longer has a record of the view (or if $S_i$ is a completely new site), the complete database must be downloaded to $S_i$. Otherwise, all the transactions which committed since the last view of which $S_i$ was a member, say view $v$, are sent to $S_i$ in the order in which they committed. This is easily done by sending that part of the log recorded since the recording of view $v$.

Thus when the view changes, if the change is to delete a site from the view, processing can continue without interruption. Committed transactions in the previous view remain committed at all sites in the view and transactions which commit in the new view will be remembered by the Logger. If the view change is an add, or both add and delete, processing of new updates must be delayed until the new sites are brought up-to-date. A site that has started a transaction, that is, performed some reads but not yet broadcast the writes when it delivers a view change message, can continue processing and broadcast the writes with the implied commit request. The request will be delivered in the next view and not until any new sites that might have joined the group are updated. Thus, the transaction need not be aborted because of the failure of one of the participants.

To summarize, recovery for a failed site, $S_i$, is this:

- Do local recovery including replay of the recovery log to undo partial effects of uncommitted transactions in the stable copy of the database and redo committed transactions not in the stable database.

- Broadcast message requesting a response from the Principal Logger for global recovery.

- Receive and process recovery information from the Logger site.

- Acknowledge to the Logger that recovery is complete.

If global recovery consists of several views, the recovering site should create a checkpoint upon completion of each view and inform the Logger of this. This way, if $S_i$ fails again before completing global recovery, the next time $S_i$ attempts global recovery, the number of views to be processed is smaller. When $S_i$ successfully completes global recovery, it acknowledges this to the Logger. The Logger then broadcasts a message to all sites telling them to begin normal operations.

### 4.2 Delayed Broadcast Recovery

The Single Broadcast protocol requires versioning of the data items so that the write-commit message can be checked for reads of stale values which will cause the transaction to be aborted. The Delayed Broadcast protocol has no such requirement and is similar to the replication protocol used by Kemme and Alonso [11]. In this paper they present an implementation using PostgreSQL of a similar protocol with two broadcasts per transaction. The protocols in this and earlier papers [10, 9] recover a failed site by doing a complete database transfer. The principles presented here for recovery using the Delayed Broadcast protocol could be applied to these similar protocols.
Type view change can be brought up-to-date and it is not aborted. by broadcasting the commit request for Items, Values various transactions since they were a part of an incomplete transaction. Thus we must abort all on-going transactions, is on-going, that is, it has broadcast and delivered the write operations but has not yet requested a view change has been deleted from the view is installed and Values since both broadcasts for each transaction will have been done in the same view.

<table>
<thead>
<tr>
<th>Record #</th>
<th>Type</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>view change</td>
<td>$v = {S_i, S_j, S_k, S_m}$</td>
</tr>
<tr>
<td>102</td>
<td>write request</td>
<td>${\text{Items, Values}}$ for $t_1$</td>
</tr>
<tr>
<td>103</td>
<td>view change</td>
<td>$v' = {S_j, S_k, S_m}$</td>
</tr>
<tr>
<td>104</td>
<td>commit request</td>
<td>$t_1$</td>
</tr>
<tr>
<td>105-114</td>
<td>write and commit</td>
<td>various transactions</td>
</tr>
<tr>
<td>115</td>
<td>write request</td>
<td>${\text{Items, Values}}$ for $t_2$</td>
</tr>
<tr>
<td>116</td>
<td>view change</td>
<td>$v'' = {S_i, S_j, S_k, S_m}$</td>
</tr>
</tbody>
</table>

Table 1: Multicast System Records Produced by Example 1

When the Delayed Broadcast protocol is used, a transaction that is on-going when a new view is installed, provides additional complication. This is because the transaction may have broadcast its writes, but has not yet broadcast the commit request when the view change occurs and in fact, may not, if it is aborted at the home site. If a transaction that originated at $S_i$, is on-going, that is, it has broadcast and delivered the write operations but has not yet requested a commit when site $S_i$ fails, $S_i$ will consider it aborted when it recovers. All correct sites should therefore abort on-going transactions from $S_i$ if $S_i$ is one of the sites deleted in the new view. What about on-going transactions from a correct site? On-going transactions from non-failed sites must also be aborted when using the multicast system’s recovery mechanism. In Example 1, Figure 1, $t_1$ is an on-going transaction from a non-failed site $S_j$ and it is not aborted.

As a result of the transactions in Example 1, the multicast system would create a message log to be replayed to $S_i$ upon recovery that looks something like Table 1. The problem here is that when site $S_i$ recovers, all of the records in the log in view $v'$ will be sent to $S_j$ and $S_j$ will get the commit request for $t_1$ but $S_j$ will not have the write operations of $t_1$ and thus cannot commit it. This is because the write operations of $t_1$ would have been lost or undone during the local recovery at $S_j$ since they were a part of an incomplete transaction. Thus we must abort all on-going transactions when a view change is delivered if we are using the member recovery mechanism of the multicast system. Thus, when $S_i$ recovers and is re-admitted to the group, the view will change again, from $v'$ to $v''$ and $S_j$ can be brought up-to-date by getting those transactions that have committed in view $v'$ since both broadcasts for each transaction will have been done in the same view.

EXAMPLE 1

View $v$ is installed
- Site $S_j$ begins transaction $t_1$
- Site $S_j$ broadcasts the writes for $t_1$ and they are delivered

View $v'$ is installed and $S_i$ has been deleted from the view
- Site $S_j$ completes transaction $t_1$ by broadcasting the commit request
- other transactions are committed
- Site $S_j$ begins transaction $t_2$
- Site $S_j$ broadcasts the writes for $t_2$ and they are delivered

View $v''$ is installed and $S_i$ rejoins the group
- The logger or recovery mechanism brings $S_i$ up-to-date by sending all messages from view $v'$
- Site $S_j$ completes transaction $t_1$ by broadcasting the commit request

Figure 1: Example 1

It is not, however, necessary to abort these on-going transactions of correct sites just because one of the participants fails when the home site is still correct if we use database sites as Loggers and make additions to the Logging protocol. There are two ways to handle the problem of Example 1. The first puts a greater processing burden on the Loggers whenever there is a view change. Since the Loggers have additional processing at each view change already, we offer
4.2.1 Log Update Method

In the first method, which we call log update, the Loggers must examine their message logs or their database state at each view change to see if there are on-going transactions at non-failed sites. If there are, the Logger should mark these transactions so that when the termination message (commit or abort) is delivered, if the message resulted in a successful commit, the Logger will find the record containing the writes for that transaction and copy it to the view change record. Thus the sequence of messages from our example produces a partial log as shown in Table 2. When view $t_1$ is installed, the Logger notes that $t_1$ is in progress. If the transaction is later aborted, the earlier writes record can be deleted and the abort request is not logged. If a commit request is received as in the case of $t_1$ (Record number 104), the Logger finds the record for the writes (which is found to be 102) and records all of the information in the view change record 103. This information in the view change record indicates previous writes to a transaction which terminates in this view. When $S_j$ rejoins the view (record number 116), the Logger begins its replay of the log by locating the appropriate view start, record 103. The information concerning “previous writes” will be sent first, followed by 104–114. Record 103 may have several “previous writes” fields corresponding to transactions which terminated during the view, and these will be replayed in the order that they were originally received, which can be determined by their record numbers.

It is possible that the earlier writes to a transaction that commits in this view are no longer in the log because all sites were active and that part of the log was garbage collected. This is not the normal case, since a transaction that multicasts write operations is usually ready to commit. However, should this happen, the Logger determines from its own database what items and values were written by the transaction so that the Logger can create a previous writes field in the view change record.

The drawback to this method is that the Loggers must keep the logs of previous views whether or not a site was missing from the group in case there were write messages for transactions that terminated in a later view.

This method is correct since the message stream is “replayed” to the recovering site in the original order, thus recreating the same conditions as the sites which have never failed. As before, the commit order is the same for all committed transactions. Effects of aborted transactions are undone at non-failed sites when they are aborted and do not affect the database state, so the fact that they are not included in the Loggers log has no effect. The only effect of the first broadcast of a transaction that is later aborted is to abort local transactions holding conflicting read locks. Since the recovering site is not initiating transactions locally, there is no effect.

Alternatively, the Logger can include the writes (that are located earlier in the log) in the commit request (in the style of the Single Broadcast message. This will produce the same result in the log as the next method and require additional processing by the lock and recovery manager as explained in the next section.

4.2.2 Augmented Broadcast (AB) Method

The second method which we call Augmented Broadcast or (AB), shifts additional processing to the home sites of on-going transactions and requires a change to the recovery lock management algorithm. If a site $S_j$ has an on-going
transaction \( t_1 \) when a new view is installed, it modifies the protocol for committing transactions so that the write requests are included in the commit request for all transactions that broadcast their writes in an earlier view. Sites which have been operational all through the view change will ignore the writes and process the commit request. Sites which are Loggers will log the augmented message. Thus, Augmented Broadcast calls for modification of the message protocol only for transactions that are in-progress when a view change occurs by adding the write information to the final commit request message. Table 3 shows what log records would be produced by the AB method for example 1.

In Example 2, Figure 3, the transactions \( t_1, t_2, \) and \( t_3 \) all write to the same data item, \( x \). This illustrates the lock management changes that must be made during recovery. Transaction \( t_2 \) was aborted because the the home site, \( S_j \), delivered the write requests of a conflicting transaction \( (t_1) \) before the write request of \( t_2 \) was delivered. Which ever one was delivered first would cause the abort of the conflicting read lock holder. The Logger will not include the writes of \( t_2 \) among the messages of view \( \nu' \).

Since \( S_j \) has broadcast the commit request for \( t_1 \), it must be the case that all local write locks were successfully obtained and are being held until the commit. Therefore when \( S_i \) receives the writes-and-commit (Augmented Broadcast) message for \( t_1 \) as part of its recovery, it must be possible to obtain the write locks and perform the operations. As both \( t_1 \) and \( t_3 \) write to data item \( x \), all sites which do not fail in \( \nu \) or \( \nu' \) will see the following operations affecting \( x \):

<table>
<thead>
<tr>
<th>Record #</th>
<th>Type</th>
<th>( v' = {S_{i2}, S_{i3}, S_{i4}, S_m} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>view change</td>
<td>( v = {S_i, S_j, S_k, S_m} ) for ( t_1 )</td>
</tr>
<tr>
<td>102</td>
<td>write request</td>
<td>{Items, Values} for ( t_1 )</td>
</tr>
<tr>
<td>103</td>
<td>view change</td>
<td>( v' = {S_j, S_k, S_m} ) for ( t_1 )</td>
</tr>
<tr>
<td>104</td>
<td>AB commit request</td>
<td>( t_1, {\text{Items, Values}} )</td>
</tr>
<tr>
<td>105-114</td>
<td>write and commit</td>
<td>various transactions for ( t_2 )</td>
</tr>
<tr>
<td>115</td>
<td>write request</td>
<td>{Items, Values} for ( t_2 )</td>
</tr>
<tr>
<td>116</td>
<td>view change</td>
<td>( v' = {S_{i2}, S_{i3}, S_{i4}, S_m} )</td>
</tr>
</tbody>
</table>

Table 3: Log Records Produced by Example 1, AB Method
EXAMPLE 2
View $v'$ is installed
- Site $S_j$ begins transaction $t_1$ and performs read($x$)
- Site $S_j$ broadcasts the writes for $t_1$
- Site $S_j$ begins transaction $t_2$ and performs read($y$)
- Site $S_j$ broadcasts the writes for $t_2$
- All sites including $S_j$ deliver the writes for $t_1$

View $v''$ is installed and $S_j$ has been deleted from the view
- All sites including $S_j$ deliver the writes for $t_2$
- Site $S_k$ broadcasts the writes for $t_3$ and they are delivered at all sites
- Site $S_j$ completes transaction $t_1$ by broadcasting the commit request augmented by the writes
- Site $S_j$ completes transaction $t_2$ by broadcasting abort request
- Site $S_k$ completes transaction $t_3$ by broadcasting the commit request augmented

Figure 3: Example 2

4.3 Broadcast Writes Recovery

The Broadcast Writes protocol is a simple extension to the popular Two Phase Locking protocols that are currently used by centralized databases. As such, it has the potential for supporting the most general transaction types in a distributed database. In the Delayed Broadcast protocol recovery, we had to consider on-going transactions when a view change occurred. However, the actual number of on-going transactions may be small or even insignificant because a transaction normally delays the broadcast of its writes until it is ready to commit. With the Broadcast Writes protocol, however, the number of on-going transactions could be large since the protocol calls for each write operation to be multicast as soon as it is requested by the transaction, rather than delaying them. It has been shown [6] that the advantage of the Broadcast Writes protocol over a replica update protocol that does not use multicast is greatest when the transactions are long with respect to the cost of broadcast. It is therefore reasonable to assume that when a view change occurs, there will be many on-going transactions and it is better not to abort all on-going transactions at each view change. Because of this, using database sites as Loggers instead of relying on the recovery mechanism provided by the multicast system could be of significant benefit.

The Augmented Broadcast global recovery method presented for the Delayed Broadcast protocol could be used for Broadcast Writes. In Augmented Broadcast, only the final broadcast for a transaction, the commit request, is affected by the need to augment with the set of write items and values. The method then works as it does for Delayed Broadcast.

When the Log Update method is used with Broadcast Writes, the Logger must be careful to remove messages from the log for a transaction that is aborted for any reason. In the case of Delayed Broadcast, only transactions that were aborted at the time of termination request had to be removed from the log. However, with the Broadcast Writes protocol, transactions can be aborted by sites because of deadlocks. If the write requests of two or more transactions cause a deadlock, all operational sites will abort one of the transactions (and the same transaction is aborted at each site). The writes of the aborted transaction are not included in the update portion of the view change record. However, the last write of the transaction to be aborted could be logged and replayed to the recovering site as in Example 3, Figure 4 if care is not taken. In this example, transactions $t_1$ and $t_2$ begin in view $v$. Site $S_4$ fails and a new view $v'$ is installed. Transaction $t_2$ performs a write operation that causes a deadlock which is detected by all sites with a deterministic detection and resolution algorithm. All sites abort $t_2$ without exchanging messages. Transaction $t_1$ then
gets committed. Now $S_j$ recovers and joins the group in view $v''$. As part of its global recovery, site $S_j$ will be instructed to perform a write of data item $x$ for transaction $t_2$ but will not know that $t_2$ should be aborted. To avoid this confusion, the messages for a transaction that is aborted for any reason must be removed from the log, not just those with an explicit abort message or whose commit request was refused. Thus, record 102 in Example 3, should be deleted.

EXAMPLE 3, Log Update

View $v$ is installed

- Site $S_j$ broadcasts then delivers write($x$) for $t_1$
- Site $S_j$ broadcasts then delivers write($y$) for $t_2$
- Site $S_j$ broadcasts then delivers write($y$) for $t_3$
  - other writes, commits, and view changes -

View $v'$ is installed and $S_j$ has been deleted from the view

- Site $S_j$ broadcasts then delivers write($x$) for $t_2$
- Site $S_j$ and other operational sites abort $t_2$
- Site $S_j$ broadcasts then delivers commit for $t_3$

View $v''$ is installed and $S_j$ rejoins the group

The logger brings $S_j$ up-to-date by sending all messages in the log from view $v'$ as follows:

Record 101: $V'$ = $\{S_j, S_k\}$
- Write($x,y$) for $t_1$
Record 102: Write($x$) for $t_2$
Record 103: Commit for $t_1$

Figure 4: Example 3, Broadcast Writes

5 Discussion and Conclusion

Several issues of global recovery remain to be explored. One issue is Logger management. For reasons of fault tolerance, there must be more than one Logger site. For simplicity, we assume there is a fixed number of Logger sites (determined in advance and based on the probability of failures). Those sites may be fixed or may change over time. If we designate one of the Loggers as the primary Logger, then the recovering site can be directed to it. However, if a new site wants to join while the first is in the recovery process, another Logger site takes over the role of primary. Such being the case, how does the new or newly recovered site know which Logger it should communicate with to complete its global update? The problem can be reduced to a problem of distributed election of which there are many solutions in the literature.

Another issue that must be dealt with is, how big will the log become and how much of the log can be garbage collected? Sites that are new to the distributed system will always require a complete database transfer. When the system is first started and no site has failed yet, there is no need to log anything when using the Single Broadcast protocol or the Augmented Broadcast method. It is only when a view change message is sent and some site has dropped out of the group that the Logger must begin logging. The Logger could store, in each view change record, the number of missing and unrecovered sites in that view. When a site recovers, requests the logged records and successfully completes global recovery, the count of missing sites can be decremented for those views used in the recovery. When the count goes to zero, the Logger knows that those records are no longer needed and can be deleted. However, there is still the problem of the site that fails and does not recover. How long does the Logger record messages in hopes that the site will recover and ask for them?

In this paper we have presented efficient solutions for global recovery for replicated database systems using one of three replica update protocols that rely on atomic multicast communication systems. These recovery methods will alleviate the problem of lengthy system unavailability and unnecessary disruption caused by server site failures.
References


